2

AVF Control Number: AVF-VSR-353.0390
89-01-30-WAN

Ada COMPILER VALIDATION SUMMARY REPORT:
Certificate Number: 900116W1.10236
Wang Laboratories, Inc.
Wang VS Ada, Version 4.1
Wang VS 8480

Completion of On-Site Testing:
16 January 1990

Prepared By:
Ada Validation Facility
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

OFFICE OF THE DIRECTOR OF
DEFENSE RESEARCH AND ENGINEERING

WASHINGTON, DC 20301

18 APR 19[illegible]

(R&AT)

MEMORANDUM FOR Director, Directorate of Database Services,
Defense Logistics Agency

SUBJECT: Technology Screening of Unclassified/Unlimited Reports

Feb 12
per [illegible]
Easter

Your letter of 2 February 1990 to the Commander, Air Force Systems Command, Air Force Aeronautical Laboratory, Wright-Patterson Air Force Base stated that the Ada Validation Summary report for Meridian Software Systems, Inc. contained technical data that should be denied public disclosure according to DoD Directive 5230.25

We do not agree with this opinion that the contents of this particular Ada Validation Summary Report or the contents of the several hundred of such reports produced each year to document the conformity testing results of Ada compilers. Ada is not used exclusively for military applications. The language is an ANSI Military Standard, a Federal Information Processing Standard, and an International Standards Organization standard. Compilers are tested for conformity to the standard as the basis for obtaining an Ada Joint Program Office certificate of conformity. The results of this testing are documented in a standard form in all Ada Validation Summary Reports which the compiler vendor agrees to make public as part of his contract with the testing facility.

On 18 December 1985, the Commerce Department issued Part 379 Technical Data of the Export Administration specifically listing Ada Programming Support Environments (including compilers) as items controlled by the Commerce Department. The AJPO complies with Department of Commerce export control regulations. When Defense Technical Information Center receives an Ada Validation Summary Report, which may be produced by any of the five U.S. and European Ada Validation Facilities, the content should be made available to the public.

If you have any further questions, please feel free to contact the undersigned at (202) 694-0209.

John P. Solomond

John P. Solomond
Director
Ada Joint Program Office

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: Wang Laboratories, Inc., Wang VS Ada, Version 4.1, Wang VS 8480 (Host & Target), 900116W1.10236	5. TYPE OF REPORT & PERIOD COVERED 16 Jan 1990 to 16 Jan 1991
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Wright-Patterson AFB Dayton, OH, USA	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Wright-Patterson AFB Dayton, OH, USA	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE
	13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Wright-Patterson AFB Dayton, OH, USA	15. SECURITY CLASS (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Wang Laboratories, Inc., Wang VS Ada, Version 4.1, Wright-Patterson AFB, Wang VS 8480 under Wang VS OS 7.29.23 (Host & Target), ACVC 1.10.

Ada Compiler Validation Summary Report:

Compiler Name: Wang VS Ada, Version 4.1

Certificate Number: 900116W1.10236

Host:	Wang VS 8480 under Wang VS OS 7.29.23
Target:	Wang VS 8480 under Wang VS OS 7.29.23

Testing Completed 16 January 1990 Using ACVC 1.10

Customer Agreement Number: 89-01-30-WAN

This report has been reviewed and is approved.

[illegible]

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

[illegible]

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

[illegible]

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

Accession For	
NTIS GRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report ~~(VSR)~~ describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability, (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation-dependent but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 16 January 1990 at Lowell MA.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C.#552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures, Version 2.0, Ada Joint Program Office, May 1989.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.

Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation of legal Ada programs with certain language constructs which cannot be verified at compile time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: Wang VS Ada, Version 4.1

ACVC Version: 1.10

Certificate Number: 900116W1.10236

Host Computer:

Machine:	Wang VS 8480
Operating System:	Wang VS OS 7.29.23
Memory Size:	16 Mbyte

Target Computer:

Machine:	Wang VS 8480
Operating System:	Wang VS OS 7.29.23
Memory Size:	16 Mbyte

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

(1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

(2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

(3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

(4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

(1) This implementation supports the additional predefined types SHORT_INTEGER, SHORT_FLOAT, and SHORT_SHORT_INTEGER in package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

(1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

(2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

(3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

(4) NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

(5) Sometimes NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

(6) Underflow is not gradual. (See tests C45524A..Z (26 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

(1) The method used for rounding to integer is round away from zero. (See tests C46012A..Z (26 tests).)

(2) The method used for rounding to longest integer is round away from zero. (See tests C46012A..Z (26 tests).)

(3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT.

For this implementation:

(1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR. (See test C36003A.)

(2) NUMERIC_ERROR is raised when an array type with INTEGER'LAST + 2 components with each component being a null array is declared. (See test C36202A.)

(3) NUMERIC_ERROR is raised when an array type with SYSTEM.MAX_INT + 2 components with each component being a null array is declared. (See test C36202B.)

(4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array type is declared. (See test C52103X.)

(5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared. (See test C52104Y.)

(6) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

(7) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

(8) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

f. Discriminated types.

(1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

(1) In the evaluation of a multi-dimensional aggregate, all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

(2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

(3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

(1) The pragma INLINE is supported for functions and procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

(1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

(2) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output.

(1) The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

(2) The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

(3) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E (2 tests), CE2102N, and CE2102P.)

(4) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I..J (2 tests), CE2102R, CE2102T, and CE2102V.)

(5) Modes IN_FILE and OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K (3 tests).)

(6) RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

(7) RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K and CE2102Y.)

(8) RESET and DELETE operations are supported for text files. (See tests CE3102F..G (2 tests), CE3104C, CE3110A, and CE3114A.)

(9) Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

(10) Temporary sequential files are given names and deleted when closed. (See test CE2108A.)

(11) Temporary direct files are given names and deleted when closed. (See test CE2108C.)

(12) Temporary text files are given names and deleted when closed. (See test CE3112A.)

(13) More than one internal file can be associated with each external file for sequential files when reading only. (See tests CE2107A..E (5 tests), CE2102L, CE2110B, and CE2111D.)

(14) More than one internal file can be associated with each external file for direct files when reading only. (See tests CE2107F..H (3 tests), CE2110D, and CE2111H.)

(15) More than one internal file can be associated with each external file for text files when reading only. (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 378 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 24 tests were required to successfully demonstrate the test objective.

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	129	1132	1945	17	26	46	3295
Inapplicable	0	6	370	0	2	0	378
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	198	577	545	245	172	99	161	332	137	36	252	261	280	3295
Inappl	14	72	135	3	0	0	5	0	0	0	0	108	41	378
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84M	CD2A84N	CD2B15C	CD2D11B	CD5007B	CD50110
ED7004B	ED7005C	ED7005D	ED7006C	ED7006D	CD7105A
CD7203B	CD7204B	CD7205C	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 378 tests were inapplicable for the reasons indicated:

a. The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)

C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

b. C35702B and B86001U are not applicable because this implementation supports no predefined type LONG_FLOAT.

c. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45614C	C45631C
C45632C	B52004D	C55B07A	B55B09C	B86001W
CD7101F				

d. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of SYSTEM.MAX_MANTISSA is less than 48.

e. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. This test recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

f. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

g. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

h. CD1009C, CD2A41A..E (5 tests), and CD2A42A..J (10 tests) are not applicable because this implementation does not support size clauses for floating point types.

i. The following 26 tests are not applicable because a length clause on a type derived from a private type is not supported outside the defining package:

CD1C04A	CD2A21C..D (2)	CD2A22C..D (2)	CD2A22G..H (2)
CD2A31C..D (2)	CD2A32C..D (2)	CD2A32G..H (2)	CD2A51C..D (2)
CD2A52C..D (2)	CD2A52G..H (2)	CD2A53D	CD2A54D
CD2A54H	CD2A72A..B (2)	CD2A75A..B (2)	

j. CD1C04B, CD1C04E, and CD4051A..D (4 tests) are not applicable because representation clauses on derived records or derived tasks are not supported.

k. The following 25 tests are not applicable because a length clause on an array or record would require change of representation of the components or elements:

CD2A61A..D (4)	CD2A61F	CD2A61H..L (5)	CD2A62A..C (3)
CD2A71A..D (4)	CD2A72C..D (2)	CD2A74A..D (4)	CD2A75C..D (2)

l. CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because this implementation does not support size clauses for access types that are less than the minimum 32 bits required.

m. The following 21 tests are not applicable because this implementation does not support an address clause for a constant:

CD5011B	CD5011D	CD5011F	CD5011H
CD5011L	CD5011N	CD5011R	CD5012C..D (2)
CD5012G..H (2)	CD5012L	CD5013B	CD5013D
CD5013F	CD5013H	CD5013L	CD5013N
CD5013R	CD5014U	CD5014W	

n. CD5012J, CD5013S, and CD5014S are not applicable because this implementation does not support an address clause for a task.

o. CE2102D is inapplicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.

p. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

q. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

r. CE2102I is inapplicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.

s. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

t. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

u. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

v. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

w. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

x. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

y. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

z. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

aa. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

ab. CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.

ac. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

ad. CE3102E is inapplicable because this implementation supports CREATE with IN_FILE mode for text files.

ae. CE3102F is inapplicable because this implementation supports RESET for text files.

af. CE3102G is inapplicable because this implementation supports deletion of an external file for text files.

ag. CE3102I is inapplicable because this implementation supports CREATE with OUT_FILE mode for text files.

ah. CE3102J is inapplicable because this implementation supports OPEN with IN_FILE mode for text files.

ai. CE3102K is inapplicable because this implementation supports OPEN with OUT_FILE mode for text files.

aj. CE2107B..E (4 tests), CE2107L, CE2110B, and CE2111D are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

ak. CE2107G..H (2 tests), CE2110D, and CE2111H are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

al. CE2401H is not applicable because CREATE with mode INOUT_FILE is not supported for unconstrained records with default discriminants.

am. EE2401D and EE2401G are not applicable because use of instantiations of DIRECT_IO with unconstrained array types and record types with discriminants with default values raise USE_ERROR without a FORM parameter.

an. CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

ao. CE3305A is not applicable because this implementation requires a bounded line length.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 24 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B23004A	B24007A	B24009A	B28003A	B28003C	B32202A
B32202B	B32202C	B33001A	B37004A	B45102A	B49003A
B49005A	B61012A	B62001B	B91004A	B95069A	B95069B
BA1101B	BC2001D	BC3009C	BD5005B		

The following tests were graded using a modified evaluation criteria:

a. BA2001E expects that the non-distinctness of names of subunits with a common ancestor be detected at compile time, but this implementation detects the errors at link time. The AVO ruled that it is also acceptable to make the error detection at link time. Thus, this test is considered to be passed if the intended errors are detected at either compile or link time.

b. EA3004D fails to detect an error because pragma INLINE is invoked for a function that is called within a package specification. By re-ordering the files, it may be shown that INLINE indeed has no effect. The AVO has ruled that the test is to be run as is, noting that it fails to detect an error where INLINE is invoked from within a package specification. The compilation files are then re-ordered. For EA3004D, the order is 0, 1, 4, 5, 2, 3, 6. The test executes and produces the expected NOT_APPLICABLE result, as though INLINE were not supported at all. The test is graded as passed.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the Wang VS Ada, Version 4.1 compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Wang VS Ada, Version 4.1 compiler using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	Wang VS 8480
Host operating system:	Wang VS OS 7.29.23
Target computer:	Wang VS 8480
Target operating system:	Wang VS OS 7.29.23
Compiler:	Wang VS Ada, Version 4.1

A set of diskettes containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the diskettes. Tests requiring modifications during the prevalidation testing were included in their modified form on the diskettes.

The contents of the diskettes were loaded onto a PC. Tests B22005Z, B22005I, B25002A, B26005A, and B27005A contain the SUB and NUL characters, which the uploading procedure treats as end-of-file. Anything in those tests occurring after those characters would be lost during the uploading process. These characters were replaced with other characters for the uploading procedure. A batch file was then used to upload the files from the PC to the host computer. Uploading was done using PCVS, a PC-to-VS transfer program which utilizes a Wang 928 communications link. Tests B22005Z, B22005I, B25002A, B26005A, and B27005A then had the SUB and NUL characters replaced.

After the test files were loaded to disk, the full set of tests was compiled, linked, and all executable tests were run on the Wang VS 8480. Results were printed from the host computer.

The compiler was tested using command scripts provided by Wang Laboratories, Inc. and reviewed by the validation team. The compiler was tested using all the following option settings. See Appendix E for a complete listing of the compiler options for this implementation. The following list of compiler options includes those options which were invoked by default:

WARNING=NO	Prevents warning messages from being included in the compilation listing.
ERRORS=999	Sets the maximum number of compilation errors permitted to 999 (compilation is terminated if this number is exceeded).
LINELEN=80	Sets the width of the listing file to 80 characters.
BANNER=NO	Prevents banners from being included in the listing file.
INLINE=YES	Allows the compiler to inline code.

Tests were compiled, linked, and executed (as appropriate) using a single computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Lowell MA and [illegible] completed on 16 January 1990.

APPENDIX A

DECLARATION OF CONFORMANCE

Wang Laboratories, Inc. has submitted the following Declaration of Conformance concerning the Wang VS Ada, Version 4.1 compiler.

DECLARATION OF CONFORMANCE

Compiler Implementor: Wang Laboratories, Inc.

Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB, OH 45433-6503

Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name:	Wang VS Ada	Version:	4.1
Host Architecture ISA:	Wang VS 8480	OS&VER #:	Wang VS OS 7.29.23
Target Architecture ISA:	Wang VS 8480	OS&VER #:	Wang VS OS 7.29.23

Implementor's Declaration

I, the undersigned, representing Wang Laboratories, Inc., have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that Wang Laboratories, Inc. is the owner of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.

[illegible] Date: 1/15/90

Wang Laboratories, Inc.
Gerald Paul, Vice President, Systems R&D

Owner's Declaration

I, the undersigned, representing Wang Laboratories, Inc., take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

[illegible] Date: 1/15/90

Wang Laboratories, Inc.
Gerald Paul, Vice President, Systems R&D

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the Wang VS Ada, Version 4.1 compiler, as described in this Appendix, are provided by Wang Laboratories, Inc. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

   ...

   type INTEGER is range -2_147_483_648 .. 2_147_483_647;
   type SHORT_INTEGER is range -32_768 .. 32_767;
   type SHORT_SHORT_INTEGER is range -128 .. 127;

   type FLOAT is digits 15 range
        -16#0.FFFF_FFFF_FFFF_FF#E+63 .. 16#0.FFFF_FFFF_FFFF_FF#E+63;
   type SHORT_FLOAT is digits 6 range -16#0.FFFF_FF#E+63 .. 16#0.FFFF_FF#E+63;

   type DURATION is delta 2#0.000_001# range -86_400.0 .. 86_400.0;

   ...

end STANDARD;
```

APPENDIX F
IMPLEMENTATION-DEPENDENT CHARACTERISTICS

This appendix describes the implementation-dependent characteristics of the VS Ada compiler. Appendix F is a required part of the LRM.

Appendix F contains the following sections:

Throughout this appendix, citations in square brackets refer to the relevant sections of the *Reference Manual for the Ada Program Language* (LRM).

F.1 IMPLEMENTATION-DEPENDENT PRAGMAS

F.1.1 Pragma INLINE

Pragma INLINE is fully supported by VS Ada with one exception: a function that is called in a declarative part cannot be expanded inline.

F.1.2 Pragma INTERFACE

Ada programs can interface to subprograms written in assembler or other languages through the predefined pragma INTERFACE [13.9] and the VS Ada-defined pragma INTERFACE_NAME. Pragma INTERFACE is described in this section. For a description of pragma INTERFACE_NAME, see Section F.1.3.

Pragma INTERFACE specifies the name of an interfaced subprogram and the name of the programming language for which calling and parameter-passing conventions are generated. The pragma takes the form specified in the LRM:

```
pragma INTERFACE (language_name, subprogram_name);
```

where

language_name is the name of the language whose calling and parameter-passing conventions are to be used;

subprogram_name is the name used within the Ada program to refer to the interfaced subprogram.

Two language names are currently accepted by pragma INTERFACE: CLE and C. The language name CLE refers to the standard VS Common Language Environment calling and parameter-passing conventions. You can use the language name CLE to interface Ada subprograms with subroutines written in any language that follows the standard VS calling conventions: BASIC, COBOL, FORTRAN, PL1, RPGII, and some C functions. You can also call Assembler programs with CLE, although there is nothing inherent in the Assembler language to expect or support this calling convention, as there is in the other languages listed.

You must use the language name C when calling VS C programs that expect the normal C calling convention. You should, however, use pragma INTERFACE CLE to call C functions that are declared with the option cle.

Calling Conventions

For both CLE and C pragma INTERFACE language names, the machine state is saved as part of the process of calling the interfaced subprogram. This is accomplished by the JSCI machine instruction, which saves the register state on the VS system stack prior to jumping to the subroutine. The Ada environment is properly restored from this save area by a normal return from the subprogram. From an Assembler subprogram, this return must be performed by the RT machine instruction.

The Ada runtime system treats any program interruption occurring during execution of the body of the interfaced subprogram as an exception being raised at the point of call of the subprogram. See Section 6.2 for a discussion of how non-Ada errors are mapped onto Ada exceptions.

Interfaced subprograms called with pragma INTERFACE may not issue any form of PCEXIT SVC. Interfaced subprograms may also be restricted from issuing any form of CEXIT SVC.

Parameter-Passing Conventions

The following conventions apply to both of the language names, CLE and C, currently accepted by pragma INTERFACE:

- Register 1 contains the address of the parameter area on entry to the subprogram.
- No consistency checking is performed between the subprogram parameters declared in Ada and the corresponding parameters of the interfaced subprogram. It is your responsibility to ensure correct access to the parameters.
- Formal parameters of mode out are not allowed.

For pragma INTERFACE CLE, register 1 contains the address of a parameter address list. Each word in this list is an address corresponding to a parameter. The most significant (sign) bit of the last word in the list is set to indicate the end of the list.

If the subprogram is a function, the result is returned in an area allocated by the Ada program and addressed by the first address in the parameter address list.

For formal parameters of mode in out, the address passed is that of the actual parameter. This provides a true pass by reference from Ada to the CLE interfaced subprogram for parameters of all types: scalar, access, record, and array. For a record type, the address in the parameter list is that of the first component of the record. For an array type, the address in the parameter list is that of the first element of the array.

Formal parameters of mode in are also all allowed for CLE interfaced subprograms. For parameters of scalar and access type, the address passed is that of a copy of the value of the actual parameter. The actual parameters are thereby protected from modification within the interfaced subprogram. For all other parameters, the address passed is the address of the actual parameter. Consequently, nonscalar and nonaccess parameters to interfaced subprograms cannot be protected from modification by the called subprogram, even when they are formally declared to be of mode in. It is your responsibility to ensure that the semantics of the Ada parameter modes are honored in these cases.

For pragma INTERFACE C, register 1 again serves to address the parameter list. These parameters are passed by value, however, in accordance with the VS C parameter-passing conventions. For all parameter types, the values of the actual parameters are passed in the parameter area.

Formal parameters of mode in are allowed for all types of actual parameters: scalar, access, record, and array. For a scalar or access parameter, the value of the actual parameter is stored in the parameter area. For a record parameter, a copy of the entire record is stored in the parameter area. For an array parameter, the address of the array is stored in the parameter area, since the value of an array in a C expression is simply the address of the first element of the first dimension of the array.

Because array parameters to C interfaced subprograms are passed by address, they cannot be protected from modification by the called subprogram, even when they are formally declared to be of mode in. It is your responsibility to ensure that the semantics of the Ada parameter mode are honored in this case.

A formal parameter of mode in out is not allowed for any scalar, access or record type but is allowed for an array type since an array parameter is passed by its address.

If the C subprogram is declared and called as a function, register 0 is used to return the result. Scalar and access values are returned in general register 0. Floating point values are returned in floating point register 0. Record values are returned by address in general register 0. Array values are not supported, since VS C does not allow a function to return an array result.

Pragma INTERFACE C lets you declare and call a C subprogram as a procedure. It is your responsibility to do so only where appropriate; specifically, when calling a C function that is declared in C to be of type VOID.

Parameter Representations

This section describes the representation of values of the types that can be passed as parameters to an interfaced subprogram. The discussion assumes that you have not used representation clauses to change the default representations of the types involved. The effect of representation clauses on the representation of values is described in Section F.4.

The following types can be passed as parameters to an interfaced subprogram:

Integer types [3.5.4] -- Ada integer types are represented in two's complement form and occupy 8 (SHORT_SHORT_INTEGER), 16 (SHORT_INTEGER), or 32 (INTEGER) bits.

Boolean types [3.5.3]-- Ada Boolean types are represented as 8-bit values: FALSE is represented by the value 0; TRUE, by the value 1.

Enumeration types [3.5.1]-- Ada enumeration types are represented internally as unsigned values corresponding to their positions in the list of enumeration literals that defines the type. The first literal in the list has the value 0.

An enumeration type can include a maximum of 2**31 values. Enumeration types with 256 elements or fewer are represented in 8 bits; those with 256 to 65536 (2**16) elements are represented in 16 bits; all others are represented in 32 bits. Accordingly, the Ada predefined type CHARACTER [3.5.2] is represented in 8 bits, using the standard ASCII codes [C].

Floating point types [3.5.7, 3.5.8] -- Ada floating point types occupy 32 (SHORT_FLOAT) or 64 (FLOAT) bits and are held in VS (short or long) format.

Fixed point types [3.5.9, 3.5.10] -- Ada fixed point types are managed by the Compiler as the product of a signed mantissa and a constant small. The mantissa is implemented as a 16- or 32-bit integer value. Small is a compile-time quantity that is the power of two equal or immediately inferior to the delta specified in the declaration of the type.

The attribute MANTISSA is defined as the smallest number such that

```
  2 ** MANTISSA >= max (abs(upper_bound), abs (lower_bound)) / small
```

The size of a fixed point type is

MANTISSA	Size
1 ..15	16 bits
16 ..31	32 bits

Fixed point types requiring a MANTISSA greater than 31 are not supported.

Access types [3.8] -- Values of Ada access types are represented internally by the 31-bit address of the designated object held in a 32-bit word. You should not alter any bits of this word, even those that are ignored by the architecture on which the program is running. The value 0 is used to represent null.

Array types [3.6] -- The elements of an array are allocated by row. When an array is passed as a parameter to an interfaced subprogram, the usual consistency checking between the array bounds declared in the calling program and the subprogram is not enforced. Therefore, it is your responsibility to ensure that the subprogram does not violate the bounds of the array.

Values of the predefined type STRING [3.6.3] are arrays and are passed in the same way as the values of any other array type. Elements of a string are represented in 8 bits, using the standard ASCII codes.

Record types [3.7] -- Components of a record are aligned on their natural boundaries (e.g., INTEGER is aligned on a word boundary), and the Compiler may reorder the components to minimize the total size of objects of the record type. If a record contains discriminants or components having a dynamic size, implicit components may be added to the record. Thus, the default layout of the internal structure of the record cannot be inferred directly from its Ada declaration. It is therefore recommended that you use a representation clause to control the layout of any record type whose values are to be passed to interfaced subprograms.

Restrictions on Interfaced Subprograms

Refer to Chapter 6 for a description of the restrictions on interfaced subprograms.

F.1.3 Pragma INTERFACE_NAME

Pragma INTERFACE_NAME associates the name of an interfaced subprogram, as declared in Ada, with its name in its language of origin. If pragma INTERFACE_NAME is not used, then the two names are assumed to be identical.

Pragma INTERFACE_NAME takes the form

```
pragma INTERFACE_NAME (subprogram_name, string_literal);
```

where

subprogram_name is the name used within the Ada program to refer to the interfaced subprogram;

string_literal is the name by which the interfaced subprogram is referred to at link time.

The use of INTERFACE_NAME is optional; you need not use it if a subprogram has the same name in Ada as it has in its language of origin. INTERFACE_NAME is useful if, for example, the name of the subprogram in its original language contains characters that are not permitted in Ada identifiers. Although Ada identifiers can contain letters, digits, and underscores, the VS Linker limits external names to the letters A-Z; the digits 0-9; and the symbols $, @, and #. Thus, only these characters are permitted in the string_literal argument of the pragma INTERFACE_NAME.

Pragma INTERFACE_NAME is allowed at the same places in an Ada program as pragma INTERFACE [13.9]. However, INTERFACE_NAME must always occur after the pragma INTERFACE declaration for the interfaced subprogram.

In order to conform to the naming conventions of the VS Linker, the link-time name of an interfaced subprogram is truncated to 32 characters and converted to upper case.

Example

```
package SAMPLE_DATA is
  function PROCESS_SAMPLE (X:INTEGER) return INTEGER;
private
  pragma INTERFACE (ASSEMBLER, PROCESS_SAMPLE);
  pragma INTERFACE_NAME (PROCESS_SAMPLE, "PSAMPLE");
end SAMPLE_DATA:
```

F.1.4 Other Pragmas

Pragmas IMPROVE and PACK -- These pragmas are discussed in detail in section F.4.

Pragma PRIORITY -- Pragma PRIORITY is accepted with the range of priorities running from 1 to 1 (see the definition of the predefined package SYSTEM, Section F.3). The undefined priority (no pragma PRIORITY) is treated as less than any defined priority value (see Appendix D).

Pragma SUPPRESS -- You can substitute the compiler option CHECKS for the pragma SUPPRESS to suppress all checks in a compilation.

The following pragmas have no effect:

- CONTROLLED
- MEMORY_SIZE
- OPTIMIZE
- STORAGE_UNIT
- SYSTEM_NAME

Note that all access types are implemented by default as controlled collections, as described in [4.8] (see Section F.10.1).

F.2 IMPLEMENTATION-DEPENDENT ATTRIBUTES

In addition to the representation attributes described in the LRM [13.7.2 and 13.7.3], VS Ada provides the following attributes:

T'VARIANT_INDEX, C'ARRAY_DESCRIPTOR, T'RECORD_SIZE, C'RECORD_DESCRIPTOR -- These attributes, which are used in record representation clauses, are described in Section F.5.

T'DESCRIPTOR_SIZE , T'IS_ARRAY -- These attributes are described in Section F.2.1.

VS Ada imposes certain limitations on the use of the attribute ADDRESS. These limitations are described in Section F.2.2.

F.2.1 The Attributes T'DESCRIPTOR_SIZE and T'IS_ARRAY

The attributes T'DESCRIPTOR_SIZE and T'IS_ARRAY are described as follows:

T'DESCRIPTOR_SIZE -- For a prefix T that denotes a type or subtype, this attribute yields the size (in bits) required to hold a descriptor for an object of the type T, allocated on the heap or written to a file. If T is constrained, T'DESCRIPTOR_SIZE yields the value 0.

T'IS_ARRAY -- For a prefix T that denotes a type or subtype, this attribute yields the value TRUE if T denotes an array type or an array subtype; otherwise, it yields the value FALSE.

F.2.2 Limitations on the Use of the Attribute ADDRESS

The attribute ADDRESS is implemented for all prefixes that have meaningful addresses. The following entities do not have meaningful addresses; each causes a compilation warning if used as a prefix to ADDRESS:

- A constant that is implemented as an immediate value; that is, one for which no space is allocated
- A package specification that is not a library unit
- A package body that is not a library unit or subunit

F.3 SPECIFICATION OF THE PACKAGE SYSTEM

```
package SYSTEM is

  type NAME is (WANG_VS);

  SYSTEM_NAME : constant NAME := NAME'FIRST;
  MIN_INT     : constant := -(2**31);
  MAX_INT     : constant := 2**31-1;
  MEMORY_SIZE : constant := 2**24;

  type ADDRESS is range MIN_INT .. MAX_INT;

  STORATE_UNIT : constant := 8;
  MAX_DIGITS   : constant := 15;
  MAX_MANTISSA : constant := 31;
  FINE_DELTA   : constant := 2#1.0#e-31;
  TICK         : constant := 0.01;
  NULL_ADDRESS : constant ADDRESS := 0;

  subtype PRIORITY is INTEGER range 1 .. 1;

  PROGRAM_CANCEL : exception;

end SYSTEM;
```

F.4 RESTRICTIONS ON REPRESENTATION CLAUSES

This section explains how objects are represented and allocated by the VS Ada Compiler and how you can use representation clauses to control the representation and allocation of objects.

As the representation of an object is closely connected with its type, this section addresses successively the representation of enumeration, integer, floating point, fixed point, access, task, array, and record types. For each class of type, the representation of the corresponding objects is described.

With the exception of array and record types, the description of each type is independent of the others. To understand the representation of an array type or of a record type, you must first understand the representation of its components.

Apart from implementation-defined pragmas, Ada provides three means to control the size of objects:

- A size specification can be used to control the size of any object.
- The predefined pragma PACK can be used to control the size of an array, an array component, a record, or a record component.
- A record representation clause can be used to control the size of a record or a record component.

The sections that follow describe the effect of size specifications on each class of type; Sections F.4.7 and F.4.8 describe the interaction among size specifications, packing, and record representation clauses on array and record types.

Representation clauses on derived record types or derived task types are not supported, and size representation clauses are not supported on types derived from private types when they are declared outside the private part of the defining package.

F.4.1 Enumeration Types

Internal codes of enumeration literals

As described in the LRM [13.3], you can use an enumeration representation clause to specify the value of the internal code associated with an enumeration literal. Enumeration representation clauses are fully implemented.

Since internal codes must be machine integers, the internal codes specified in an enumeration representation clause must be in the range -2^{31} .. $2^{31}-1$.

If you do not specify an enumeration representation clause, the internal code associated with an enumeration literal is the position number of the enumeration literal. Thus, for an enumeration type with n elements, the internal codes are the integers 0, 1, 2, .. n-1.

Encoding of enumeration values

In the program generated by the Compiler, an enumeration value is always represented by its internal code.

Minimum size of an enumeration subtype

The minimum size of an enumeration subtype is the minimum number of bits necessary to represent the internal codes of the subtype values in normal binary form.

If a static subtype has a null range, its minimum size is 1. Otherwise, if m and M are the values of the internal codes associated with the first and last enumeration values of the subtype, then its minimum size L is determined as follows:

For $m >= 0$, L is the smallest positive integer such that $M <= 2^{L}-1$.
For $m < 0$, L is the smallest positive integer such that $-2^{L-1} <= m$ and $M <= 2^{L-1}-1$.

For example:

```
type COLOR is (GREEN, BLACK, WHITE, RED, BLUE, YELLOW);
-- The minimum size of COLOR is 3 bits.

subtype BLACK_AND_WHITE is COLOR range BLACK .. WHITE;
-- The minimum size of BLACK_AND_WHITE is 2 bits.

subtype BLACK_OR_WHITE is BLACK_AND_WHITE range X .. X;
-- Assuming that X is not static, the minimum size of BLACK_OR_WHITE
-- is 2 bits (the same as the minimum size of the static type mark
-- BLACK_AND_WHITE).
```

Size of an enumeration subtype

You can specify the size of an enumeration type and each of its subtypes in the length clause of a size specification. The length clause also determines the size of a first-named subtype. You must of course specify a value greater than or equal to the minimum size of the type or subtype.

For example:

```
 type EXTENDED is
   ( -- The usual American ASCII characters.
      NUL,  SOH,  STX,  ETX,  EOT,  ENQ,  ACK,  BEL,
      BS,   HT,   LF,   VT,   FF,   CR,   SO,   SI,
      DLE,  DC1,  DC2,  DC3,  DC4,  NAK,  SYN,  ETB,
      CAN,  EM,   SUB,  ESC,  FS,   GS,   RS,   US,
      ' ',  '!',  '"',  '#',  '$',  '%',  '&',  ''',
      '(',  ')',  '*',  '+',  ',',  '-',  '.',  '/',
      '0',  '1',  '2',  '3',  '4',  '5',  '6',  '7',
      '8',  '9',  ':',  ';',  '<',  '=',  '>',  '?',
      '@',  'A',  'B',  'C',  'D',  'E',  'F',  'G',
      'H',  'I',  'J',  'K',  'L',  'M',  'N',  'O',
      'P',  'Q',  'R',  'S',  'T',  'U',  'V',  'W',
      'X',  'Y',  'Z',  '[',  '\',  ']',  '^',  '_',
      '`',  'a',  'b',  'c',  'd',  'e',  'f',  'g',
      'h',  'i',  'j',  'k',  'l',  'm',  'n',  'o',
      'p',  'q',  'r',  's',  't',  'u',  'v',  'w',
      'x',  'y',  'z',  '{',  '|',  '}',  '~',  DEL,
      -- Extended characters
      LEFT_ARROW,
      RIGHT_ARROW,
      UPPER_ARROW,
      LOWER_ARROW,
      UPPER_LEFT_CORNER,
      UPPER_RIGHT_CORNER,
      LOWER_RIGHT_CORNER,
      LOWER_LEFT_CORNER,
      ...);

 for EXTENDED'SIZE use 8;
 -- The size of type EXTENDED will be one byte.  Its objects will be
 -- represented as unsigned 8-bit integers.
```

The Compiler fully implements size specifications. As enumeration values are represented as integers, however, the length you specify cannot be greater than 32 bits, the size of the largest predefined integer on the VS.

If you do not use a size specification for an enumeration type or first-named subtype, the objects of that type or subtype are represented as signed integers if the internal code associated with the first enumeration value is negative and as unsigned integers otherwise. VS Ada provides 8-, 16- and 32-bit integers; the Compiler automatically selects the smallest machine integer that can hold each of the internal codes of the enumeration type or subtype. Thus, the size of the enumeration type and any of its subtypes is 8, 16 or 32 bits.

Size of the objects of an enumeration subtype

Provided it is not constrained by a record component clause or a pragma PACK, an object of an enumeration subtype is the same size as its subtype.

Alignment of an enumeration subtype

An enumeration subtype is byte-aligned if the size of the subtype is less than or equal to 8 bits, halfword-aligned if the size of the subtype is less than or equal to 16 bits, and word-aligned otherwise.

Address of an object of an enumeration subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an enumeration subtype is a multiple of the alignment of the corresponding subtype.

F.4.2 Integer Types

Predefined integer types

The VS provides three predefined integer types:

```
  type SHORT_SHORT_INTEGER is range -2**07 .. 2**07-1;
  type SHORT_INTEGER       is range -2**15 .. 2**15-1;
  type INTEGER             is range -2**31 .. 2**31-1;
```

Selection of the parent of an integer type

An integer type declared as

```
  type T is range L .. R;
```

is implicitly derived from either the SHORT_INTEGER or INTEGER predefined integer type. The Compiler automatically selects the predefined integer type with the shortest range containing the values L to R inclusive. Note that the the Compiler never automatically selects the SHORT_SHORT_INTEGER representation.

Encoding of integer values

In the program generated by the Compiler, integer values are represented in binary code in a conventional two's complement form.

Minimum size of an integer subtype

The minimum size of an integer subtype is the minimum number of bits necessary to represent the internal codes of the subtype values in normal binary form (that is, in an unbiased form that includes a sign bit only if the range of the subtype includes negative values).

If a static subtype has a null range, its minimum size is 1. Otherwise, if m and M are the lower and upper bounds of the subtype, then its minimum size L is determined as follows:

For $m >= 0$, L is the smallest positive integer such that $M <= 2^{L}-1$; for $m < 0$, L is the smallest positive integer such that $-2^{L-1} <= m$ and $M <= 2^{L-1}-1$.

For example:

```
  subtype S is INTEGER range 0 .. 7;
  -- The minimum size of S is 3 bits.

  subtype D is S range X .. Y;
  -- Assuming that X and Y are not static, the minimum size of
  -- D is 3 bits (the same as the minimum size of the static type
  -- mark S).
```

Size of an integer subtype

The sizes of the predefined integer types SHORT_SHORT_INTEGER, SHORT_INTEGER, and INTEGER are 8, 16 and 32 bits, respectively.

You can specify the size of an integer type and each of its subtypes in the length clause of a size specification. The length clause also determines the size of a first-named subtype. You must of course specify a value greater than or equal to the minimum size of the type or subtype.

For example:

```
  type S is range 80 .. 100;
  for S'SIZE use 32;
  -- S is derived from SHORT_INTEGER, but its size is 32 bits
  -- because of the size specification.

  type J is range 0 .. 255;
  for J'SIZE use 8;
  -- J is derived from SHORT_INTEGER, but its size is 8 bits because
  -- of the size specification.

  type N is new J range 80 .. 100;
  -- N is indirectly derived from SHORT_INTEGER, but its size is 8 bits
  -- because N inherits the size specificaticn of J.
```

The Compiler implements size specifications. As integers are implemented using machine integers, however, the length specified cannot be greater than 32 bits.

If you do not use a size specification for an integer type or its first-named subtype (if any), the size of the integer and any of its subtypes is the size of the predefined type from which it derives, directly or indirectly.

For example:

```
  type S is range 80 .. 100;
  -- S is derived from SHORT_INTEGER; its size is 16 bits.

  type J is range 0 .. 65535;
  -- J is derived from INTEGER; its size is 32 bits.

  type N is new J range 80 .. 100;
  -- N is indirectly derived from INTEGER; its size is 32 bits.
```

Size of the objects of an integer subtype

Provided it is not constrained by a record component clause or a pragma PACK, an object of an integer subtype is the same size as its subtype.

Alignment of an integer subtype

An integer subtype is byte-aligned if the size of the subtype is less than or equal to 8 bits, halfword-aligned if the size of the subtype is less than or equal to 16 bits, and word-aligned otherwise.

Address of an object of an integer subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an integer subtype is a multiple of the alignment of the corresponding subtype.

F.4.3 Floating Point Types

Predefined floating point types

VS Ada provides two predefined floating point types:

```
 type SHORT_FLOAT is
      digits 6 range -2.0**252 *(1.0-2.0**-24) .. 2.0**252* (1.0-2.0**-24)

 type FLOAT is
      digits 15 range -2.0**252 *(1.0-2.0**-56) ..2.0**252* (1.0-2.0**-56)
```

Selection of the parent of a floating point type

A floating point type declared as

```
  type T is digits D [range L .. R];
```

is implicitly derived from a predefined floating point type. The Compiler automatically selects the smallest predefined floating point type whose number of digits is greater than or equal to D and that contains the values L and R.

Encoding of floating point values

In the program generated by the Compiler, floating point values are represented using VS data formats for single-precision or double-precision floating point values, as appropriate.

Values of the predefined type SHORT_FLOAT are represented using the single precision format; values of the predefined type FLOAT are represented using the double precision format. The values of any other floating point type are represented in the same format as the values of the predefined type from which it derives, directly or indirectly.

Minimum size of a floating point subtype

The minimum size of a floating point subtype is 32 bits if its base type is SHORT_FLOAT or a type derived from SHORT_FLOAT and 64 bits if its base type is FLOAT or a type derived from FLOAT.

Size of a floating point subtype

The sizes of the predefined floating point types SHORT_FLOAT and FLOAT are 32 and 64 bits, respectively.

The size of a floating point type and the size of any of its subtypes is the size of the predefined type from which it derives, directly or indirectly.

The only size you can specify for a floating point type or first-named subtype in a size specification is its usual size (32 or 64 bits).

Size of the objects of a floating point subtype

An object of a floating point subtype has the same size as its subtype.

Alignment of a floating point subtype

A floating point subtype is word-aligned if its size is 32 bits and double word-aligned otherwise.

Address of an object of a floating point subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a floating point subtype is a multiple of the alignment of the corresponding subtype.

F.4.4 Fixed Point Types

Small of a fixed point type

You can specify the value of small in the length clause of a size specification. The value you specify must be a power of two.

If you do not use a size specification to specify small of a fixed point type, then the value of small is determined by the value of delta, as defined by the LRM [3.5.9].

Predefined fixed point types

VS Ada provides a set of anonymous predefined fixed point types of the form

```
  type FIXED is delta D range (-2**15-1)*S .. (2**15)*S;
  for FIXED'SMALL use S;

  type LONG_FIXED is delta D range (-2**31-1)*S .. (2**31)*S;
  for LONG_FIXED'SMALL use S;
```

where D is any real value and S is any power of two less than or equal to D.

Selection of the parent of a fixed point type

A fixed point type declared as

```
  type T is delta D range L .. R;
```

optionally, with a small specification

```
  for T'SMALL use S;
```

is implicitly derived from a predefined fixed point type. The Compiler automatically selects the predefined fixed point type whose small and delta are the same as the small and delta of T and whose range is the shortest that includes the values L and R.

Encoding of fixed point values

In the program generated by the Compiler, a safe value V of fixed point subtype F is represented as the integer

```
V / F'BASE'SMALL
```

Minimum size of a fixed point subtype

The minimum size of a fixed point subtype is the minimum number of binary digits necessary to represent the values of the range of the subtype using the small of the base type (that is, in an unbiased form that includes a sign bit only if the range of the subtype includes negative values).

If a static subtype has a null range, its minimum size is 1. Otherwise, if s and S are the bounds of the subtype, and if i and I are the integer representations of m and M (the smallest and greatest model numbers of the base type such that s < m and M < S), then the minimum size L is determined as follows:

For i >= 0, L is the smallest positive integer such that I <= 2^{L}-1; for i < 0, L is the smallest positive integer such that -2^{L-1} <= i and I <= 2^{L-1}-1.

For example:

```
type F is delta 2.0 range 0.0 .. 500.0;
-- The minimum size of F is 8 bits.

subtype S is F delta 16.0 range 0.0 .. 250.0;
-- The minimum size of S is 7 bits.

subtype D is S range X .. Y;
-- Assuming that X and Y are not static, the minimum size of D is
-- 7 bits (the same as the ~~minimum size~~ of its type mark S).
```

Size of a fixed point subtype

The sizes of the sets of predefined fixed point types FIXED and LONG_FIXED are 16 and 32 bits, respectively.

You can specify the size of a fixed point type and each of its subtypes in the length clause of a size specification. The length clause also determines the size of a first-named subtype. You must of course specify a value greater than or equal to the minimum size of the type or subtype.

For example:

```
  type F is delta 0.01 range 0.0 .. 2.0;
  for F'SIZE use 32;
  -- F is derived from a 16 bit predefined fixed type, but its size is
  -- 32 bits because of the size specification.

  type L is delta 0.01 range 0.0 .. 300.0;
  for F'SIZE use 16;
  -- F is derived from a 32 bit predefined fixed type, but its size is
  -- 16 bits because of the size specification.  The size
  -- specification is legal since the range contains no negative values
  -- and therefore no sign bit is required.

  type N is new F range 0.8 .. 1.0;
  -- N is indirectly derived from a 16 bit predefined fixed type, but
  -- its size is 32 bits because N inherits the size specification of
  -- F.
```

The VS Ada Compiler implements size specifications. As fixed point objects are represented using machine integers, however, the length specified cannot be greater than 32 bits.

If you do not use a size specification for a fixed point type or its first-named subtype, the size of the fixed point type and any of its subtypes is the size of the predefined type from which it derives, directly or indirectly.

For example:

```
  type F is delta 0.01 range 0.0 .. 2.0;
  -- F is derived from a 16 bit predefined fixed type; its size is
  -- 16 bits.

  type L is delta 0.01 range 0.0 .. 300.0;
  -- L is derived from a 32 bit predefined fixed type; its size is
  -- 32 bits.

  type N is new L range 0.0 .. 2.0;
  -- N is indirectly derived from a 32 bit predefined fixed type; its
  -- size is 32 bits.
```

Size of the objects of a fixed point subtype

Provided it is not constrained by a record component clause or a pragma PACK, an object of a fixed point type is the same size as its subtype.

Alignment of a fixed point subtype

A fixed point subtype is byte-aligned if its size is less than or equal to 8 bits, halfword-aligned if the size of the subtype is less than or equal to 16 bits, and word-aligned otherwise.

Address of an object of a fixed point subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a fixed point subtype is a multiple of the alignment of the corresponding subtype.

F.4.5 Access Types

Collection Size

As described in the LRM [13.2], you can use the length clause of a size specification to indicate the amount of storage space to be reserved for the collection of an access type. The Compiler fully implements this kind of specification.

If you do not specify collection size for an access type, no storage space is reserved for its collection; the value of the attribute STORAGE_SIZE is then 0.

Minimum size of an access subtype

The minimum size of an access subtype is 32 bits.

Size of an access subtype

The size of an access subtype is 32 bits, the same as its minimum size.

The only size you can specify for an access type in a size specification is its usual size (32 bits).

Size of the objects of an access subtype

An object of an access subtype is the same size as its subtype; thus, an object of an access subtype is always 32 bits.

Alignment of an access subtype

An access subtype is always word-aligned.

Address of an object of an access subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an access subtype is always on a word boundary since its subtype is word-aligned.

F.4.6 Task Types

Storage for a task activation

As described in the LRM [13.2], you can use the length clause of a size specification to indicate the amount of storage space to be reserved for the activation of every task of a given type.

If you do not specify storage space in a length clause, the amount of storage space you indicate at bind time is allocated.

You cannot use a length clause with a derived type. VS Ada reserves the same amount of storage space for the activation of a task of a derived type as for the activation of a task of the parent type.

Encoding of task values

Task values are machine addresses.

Minimum size of a task subtype

The minimum size of a task subtype is 32 bits.

Size of a task subtype

The size of a task subtype is 32 bits, the same as its minimum size.

The only size you can specify for a task type in a size specification is its usual size (32 bits).

Size of the objects of a task subtype

An object of a task subtype is the same size as its subtype. Thus, an object of a task subtype is always 32 bits.

Alignment of a task subtype

A task subtype is always word-aligned.

Address of an object of a task subtype

Provided its alignment is not constrained by a record representation clause, the address of an object of a task subtype is always on a word boundary since its subtype is word-aligned.

F.4.7 Array Types

Layout of an array

Every array is allocated in a contiguous area of storage units. All the components of an array are the same size. A gap may exist between two consecutive components and after the last component. All gaps are the same size.

Components

If the array is not packed, its components are the same size as the subtype of the components.

For example:

```
  type A is array (1 .. 8) of BOOLEAN;
  -- The size of the components of A is the size of the
  -- type BOOLEAN: 8 bits.

  type DECIMAL_DIGIT is range 0 .. 9;
  for DECIMAL_DIGIT'SIZE use 4;
  type BINARY_CODED_DECIMAL is
    array (INTEGER range <>) of DECIMAL_DIGIT;
  -- The size of the type DECIMAL_DIGIT is 4 bits.  Thus, in an array
  -- of type BINARY_CODED_DECIMAL, each component will be represented
 -- in 4 bits as in the usual BCD representation.
```

If the array is packed and its components are neither records nor arrays, the size of the components is the minimum size of the subtype of the components.

For example:

```
type A is array (1 .. 8) of BOOLEAN;
pragma PACK(A);
-- The size of the components of A is the minimum size of the type
-- BOOLEAN:  1 bit.

type DECIMAL_DIGIT is range 0 .. 9;
type BINARY_CODED_DECIMAL is
     array (INTEGER range <>) of DECIMAL_DIGIT;
pragma PACK(BINARY_CODED_DECIMAL);
-- The size of the type DECIMAL_DIGIT is 16 bits; but as
-- BINARY_CODED_DECIMAL is packed, each component of an array of this
-- type will be represented in 4 bits as in the usual BCD
-- representation.
```

Packing the array has no effect on the size of its components when the components are records or arrays.

Gaps

Provided an array is not packed, its components are records or arrays, and no size specification applies to the subtype of the components, the Compiler may choose a representation with a gap after each component. By inserting such gaps, the Compiler optimizes access to the array components and their subcomponents. The size of the gaps is such that the relative displacement of consecutive components is a multiple of the alignment of the subtype of the components. This strategy gives each component and subcomponent an address consistent with the alignment of its subtype

For example:

```
type R is
     record
          K : INTEGER; -- INTEGER is word aligned.
          B : BOOLEAN; -- BOOLEAN is byte aligned.
     end record;
-- Record type R is word-aligned.  Its size is 40 bits.

type A is array (1 .. 10) of R;
-- A gap of three bytes is inserted after each component in order to
-- respect the alignment of type R.  The size of an array of type A
-- will be 640 bits.
```

If the array is packed, or if a size specification does apply to the subtype of the components, no gaps are inserted.

For example:

```
type R is
     record
          K : INTEGER;
          B : BOOLEAN;
     end record;

type A is array (1 .. 10) of R;
pragma PACK(A);
-- There is no gap in an array of type A because A is packed.
-- The size of an object of type A will be 400 bits.

type NR is new R;
for NR'SIZE use 40;

type B is array (1 .. 10) of NR;
-- There is no gap in an array of type B because NR has a
-- size specification.
-- The size of an object of type B will be 400 bits.
```

Size of an array subtype

The size of an array subtype is obtained by multiplying the number of its components by the sum of the size of the components and the size of any gaps. If the subtype is unconstrained, the maximum number of components is used to determine the size.

The size of an array subtype cannot be computed at compile time in the following cases:

- If the array subtype has nonstatic constraints or if it is an unconstrained array type with nonstatic index subtypes (because the number of components can then only be determined at run time).

- If the components are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static (because the size of the components and the size of the gaps can then only be determined at run time).

As indicated previously, the effect of a pragma PACK on an array type is to suppress the gaps and reduce the size of the components. Thus, packing an array type reduces its size.

If the components of an array are records or arrays and their constraints or the constraints of any subcomponents are not static, the Compiler ignores any pragma PACK applied to the array type but issues a warning message. Apart from this limitation, array packing is fully implemented by the VS Ada Compiler.

The only size you can specify for an array type or first-named subtype in a the length clause of a size specification is its usual size. Nevertheless, a length clause can be useful to verify that the layout of an array is the layout expected by the application.

Size of the objects of an array subtype

An object of an array subtype is always the same size as the subtype of the object.

Alignment of an array subtype

If no pragma PACK applies to an array subtype and no size specification applies to its components, the array subtype has the same alignment as the subtype of its components.

If a pragma PACK does apply to an array subtype or if a size specification applies to its components (so that there are no gaps), the alignment of the array subtype is the lesser of the alignment of the subtype of its components and the relative displacement of the components.

Address of an object of an array subtype

Provided its alignment is not constrained by a record representation clause, the address of an object of an array subtype is a multiple of the alignment of the corresponding subtype.

F.4.8 Record Types

Layout of a record

Every record is allocated in a contiguous area of storage units. The size of a record component depends on its type. Gaps may exist between some components.

As described in the LRM [13.4], you can use a record representation clause to control the positions and sizes of a record's components. VS Ada imposes no restrictions on a component's position. Bits within a storage unit are numbered from 0 to 7, with the most-significant bit numbered 0. The range of bits specified in a component clause may extend into following storage units. If a component is not a record or an array, its size can be any size from the minimum size to the size of its subtype. If a component is a record or an array, its size must be the size of its subtype:

```
type CONDITIONS is (ZERO, LESS_THAN, GREATER_THAN, OVERFLOW);
-- The size of CONDITIONS is 8 bits; the minimum size is 2 bits

type PROG_EXCEPTION is (FIX_OVFL, DEC_OVFL, EXP_UNDFL, SIGNIF);
type PROG_MASK is array (PROG_EXCEPTION) of BOOLEAN;
pragma PACK (PROG_MASK);
-- The size of PROG_MASK is 4 bits

type ADDRESS is range 0..2**24-1;
for ADDRESS'SIZE use 24;
-- ADDRESS represents a 24-bit memory address

type INTERRUPT is (IO, CLOCK, MACHINE_CHECK);
type INTERRUPT_MASK_TYPE is array (INTERRUPT) of BOOLEAN;

type INTERRUPT_CODE is range 0 .. 7;

type PROC_LEVEL is range 0 .. 7;
type PROGRAM_CONTROL_WORD is
  record
      INTERRUPT_CAUSE          : INTERRUPT_CODE;
      CURRENT_INST_ADDRESS     : ADDRESS;
      WAIT_STATE               : BOOLEAN;
      CONTROL_MODE             : BOOLEAN;
      PROTECTION_TRAP          : BOOLEAN:
      VIRTUAL_MACHINE          : BOOLEAN;
      INTERRUPT_MASK           : INTERRUPT_MASK_TYPE;
      DEBUG_CONTROL            : BOOLEAN;
      CONDITION_CODE           : CONDITIONS;
      PROGRAM_MASK             : PROG_MASK;
      PROCESS_LEVEL            : PROC_LEVEL;
  end record;

-- This type can be used to map the program control word of the VS.

for PROGRAM_CONTROL_WORD use
  record at mod 4;
      INTERRUPT_CAUSE          at 0    range 0 .. 7;
      CURRENT_INST_ADDRESS     at 1    range 8 .. 31;
      WAIT_STATE               at 4    range 0 .. 0;
      CONTROL_MODE             at 4    range 1 .. 1;
      PROTECTION_TRAP          at 4    range 2 .. 2;
      VIRTUAL_MACHINE          at 4    range 3 .. 3;
      INTERRUPT_MASK           at 4    range 5 .. 7;-- bit 4 unused
      DEBUG_CONTROL            at 5    range 0 .. 0;-- bits 1..7 unused
      CONDITION_CODE           at 6    range 0 .. 1;
      PROGRAM_MASK             at 6    range 2 .. 5;-- bits 6,7 unused
      PROCESS_LEVEL            at 7    range 5 .. 7;-- bits 0..4 unused
  end record;
```

You need not specify the size and position of every component of a record in a record representation clause. If you do not specify size, the component is the same size as its subtype. If you do not specify position, the Compiler chooses the position that optimizes access to the components of the record: the offset of the component is a multiple of the alignment of the component subtype. The Compiler also chooses a position that reduces the number of gaps and thus the size of the record objects.

Because of these optimizations, there is no connection between the order of components in a record type declaration and the positions of the components in a record object.

Pragma PACK has no further effect on records. The Compiler always optimizes the layout of records as described above.

The current version of VS Ada does not allow you to use a record representation clause with a derived type. The same storage representation is used for an object of a derived type as for an object of the parent type.

Indirect components

If the offset of a component cannot be computed at compile time, the offset is stored in the record objects at run time and used to access the component. Such a component is said to be *indirect;* other components are said to be *direct.*

If a record component is a record or an array, the size of its subtype may be evaluated at run time and may even depend on the discriminants of the record. We will call these components *dynamic* components.

For example:

```
  type DEVICE is (SCREEN, PRINTER);

  type COLOR is (GREEN, RED, BLUE);

  type SERIES is array (POSITIVE range <>) of INTEGER;

  type GRAPH (L : NATURAL) is
       record
            X : SERIES(1 .. L); -- The size of X depends on L
            Y : SERIES(1 .. L); -- The size of Y depends on L
       end record;

  Q : POSITIVE;
```

```
type PICTURE (N : NATURAL; D : DEVICE) is
     record
          F : GRAPH(N); -- The size of F depends on N
          S : GRAPH(Q); -- The size of S depends on Q
          case D is
               when SCREEN =>
                    C : COLOR;
               when PRINTER =>
                    null;
          end case;
     end record;
```

Any component placed after a dynamic component has an offset that cannot be evaluated at compile time and is thus indirect. In order to minimize the number of indirect components, the Compiler groups the dynamic components together and places them at the end of the record.

Thus, the only indirect components are dynamic components. But not all dynamic components are necessarily indirect: If a component list that is not followed by a variant part contains dynamic components, exactly one dynamic component of this list is a direct component because its offset can be computed at compilation time.

The offset of an indirect component is always expressed in storage units.

The space reserved for the offset of an indirect component must be large enough to store the size of any value of the record type (the maximum potential offset). The Compiler evaluates an upper bound MS of this size and treats an offset as a component having an anonymous integer type whose range is 0 .. MS.

If C is *the name of an indirect component, then the offset of* this component can be denoted in a component clause by the implementation-generated name C'OFFSET.

Implicit components

In some circumstances, access to an object of a record type or to its components involves computing information that depends only on the discriminant values. To avoid unnecessary recomputation, the Compiler stores this information in the record objects, updates it when the values of the discriminants are modified, and uses it when the objects or their components are accessed. This information is stored in special components called *implicit* components.

An implicit component may contain information that is used when the record object or several of its components are accessed. In this case, the component is included in any record object; that is, the implicit component is considered to be declared before any variant part in the record type declaration. There are two components of this kind: RECORD_SIZE and VARIANT_INDEX.

On the other hand, an implicit component may be used to access a given record component. In this case the implicit component exists whenever the record component exists; that is, the implicit component is considered to be declared at the same place as the record component. There are two components of this kind: ARRAY_DESCRIPTORS and RECORD_DESCRIPTORS.

The implicit components RECORD_SIZE, VARIANT_INDEX, ARRAY_DESCRIPTORS, and RECORD_DESCRIPTORS are described as follows:

RECORD_SIZE - The Compiler creates the implicit component RECORD_SIZE when the record type has a variant part and its discriminants are defaulted. RECORD_SIZE contains the size of the storage space needed to store the current value of the record object. (Note that the storage actually allocated for the record object may be larger.)

The value of a RECORD_SIZE component may denote a number of storage units or a number of bits. Generally it denotes a number of storage units, but if any component clause specifies that a component of the record type has an offset or a size that cannot be expressed using storage units, then the value denotes a number of bits.

The implicit component RECORD_SIZE must be large enough to store the maximum size of any value of the record type. The Compiler evaluates an upper bound MS of this size and then considers the implicit component to have an anonymous integer type whose range is 0 .. MS.

If R is the name of the record type, the implicit component RECORD_SIZE can be denoted in a component clause by the implementation-generated name R'RECORD_SIZE.

VARIANT_INDEX -- The Compiler creates the implicit component VARIANT_INDEX when the record type has a variant part. VARIANT_INDEX indicates the set of components that are present in a record value. It is used when a discriminant check is to be done.

Component lists that do not contain a variant part are numbered. These numbers are the possible values of the implicit component VARIANT_INDEX.

For example:

```
type VEHICLE is (AIRCRAFT, ROCKET, BOAT, CAR);

type DESCRIPTION (KIND : VEHICLE := CAR) is
     record
          SPEED : INTEGER;
          case KIND is
               when AIRCRAFT | CAR =>
                    WHEELS : INTEGER;
                    case KIND is
                         when AIRCRAFT => -- 1
                              WINGSPAN : INTEGER;
                         when others =>    -- 2
                              null;
                    end case;
               when BOAT =>          -- 3
                    STEAM : BOOLEAN;
               when ROCKET =>        -- 4
                    STAGES : INTEGER;
          end case;
     end record;
```

The value of the variant index indicates the set of components that are present in a record value:

Variant Index	Set
1	{KIND, SPEED, WHEELS, WINGSPAN}
2	{KIND, SPEED, WHEELS}
3	{KIND, SPEED, STEAM}
4	{KIND, SPEED, STAGES}

A comparison between the variant index of a record value and the bounds of an interval serves to check that a given component is present in the value:

Component	Interval
KIND	--
SPEED	--
WHEELS	1 .. 2
WINGSPAN	1 .. 1
STEAM	3 .. 3
STAGES	4 .. 4

The implicit component VARIANT_INDEX must be large enough to store the number V of component lists that do not contain variant parts. The Compiler treats this implicit component as having an anonymous integer type whose range is 1 .. V.

If R is the name of the record type, VARIANT_INDEX can be specified in a component clause by the implementation-generated name R'VARIANT_INDEX.

ARRAY_DESCRIPTOR -- The Compiler associates the implicit component ARRAY_DESCRIPTOR with each record component whose subtype is an anonymous array subtype that depends on a discriminant of the record. ARRAY_DESCRIPTOR contains information about the component subtype.

The structure of the implicit component ARRAY_DESCRIPTOR is not described in this documentation. However, if you wish to specify the location of an ARRAY_DESCRIPTOR component in a component clause, you can obtain the size of the component by specifying DMAP = YES when you run the Compiler.

The Compiler treats an ARRAY_DESCRIPTOR implicit component as having an anonymous record type. If C is the name of the record component whose subtype is described by the array descriptor, then ARRAY_DESCRIPTOR can be specified in a component clause by the implementation-generated name C'ARRAY_DESCRIPTOR.

RECORD_DESCRIPTOR -- The Compiler associates the implicit component RECORD_DESCRIPTOR with each record component whose subtype is an anonymous record subtype that depends on a discriminant of the record. RECORD_DESCRIPTOR contains information about the component subtype.

The structure of the implicit component RECORD_DESCRIPTOR is not described in this documentation. However, if you wish to specify the location of a RECORD_DESCRIPTOR component in a component clause, you can obtain the size of the component by specifying DMAP = YES when you run the Compiler.

The Compiler treats a RECORD_DESCRIPTOR implicit component as having an anonymous record type. If C is the name of the record component whose subtype is described by the record descriptor, then RECORD_DESCRIPTOR can be specified in a component clause by the implementation-generated name C'RECORD_DESCRIPTOR.

Suppressing implicit components

The VS Ada-defined pragma IMPROVE enables you to suppress the implicit components RECORD_SIZE and/or VARIANT_INDEX from a record type. The syntax of pragma IMPROVE is as follows:

```
pragma IMPROVE ( TIME | SPACE , [ON =>] simple_name );
```

The first argument specifies whether TIME or SPACE is the primary criterion for the choice of the representation of the record type, which is denoted by the second argument.

If you specify TIME, the Compiler inserts implicit components as described above. If you specify SPACE, the Compiler inserts a VARIANT_INDEX or a RECORD_SIZE component only if that component appears in a record representation clause that applies to the record type. Thus, you can use a record representation clause to keep one implicit component while suppressing the other.

A pragma IMPROVE that applies to a given record type can occur anywhere a representation clause is allowed for that type.

Size of a record subtype

Unless a component clause specifies that a component of a record type has an offset or a size which cannot be expressed using storage units, the size of a record subtype is rounded up to a whole number of storage units.

The size of a constrained record subtype is obtained by adding the sizes of its components and the sizes of any gaps. The size is not computed at compile time if the record subtype has nonstatic constraints or if a component is an array or a record and its size is not computed at compile time.

The size of an unconstrained record subtype is obtained by adding the sizes of the components and the sizes of any gaps of its largest variant. If the size of a component or gap cannot be evaluated exactly at compile time, the Compiler uses an upper bound of this size to compute the subtype size.

A size specification applied to a record type or first-named subtype has no effect: The only size you can specify is the default size of the record type or first-named subtype. Nevertheless, a length clause can be useful to verify that the layout of a record is the layout expected by the application.

Size of the objects of a record subtype

An object of a constrained record subtype is the same size as its subtype.

An object of an unconstrained record subtype is the same size as its subtype if that size is less than or equal to 8 Kbyte. If the size of the subtype is greater than 8 Kbyte, the object is the size that is necessary to store its current value. Storage space is allocated and released as the discriminants of the record change.

Alignment of a record subtype

When no record representation clause applies to its base type, a record subtype has the same alignment as the component with the highest alignment requirement.

When a record representation clause that does not contain an alignment clause applies to its base type, a record subtype has the same alignment as the component with the highest alignment requirement that has not been overridden by its component clause.

When a record representation clause that contains an alignment clause applies to its base type, a record subtype has the alignment specified by the alignment clause.

Address of an object of a record subtype

Provided its alignment is not constrained by a representation clause, the address of an object of a record subtype is a multiple of the alignment of the corresponding subtype.

F.5 Conventions for Implementation-Generated Names

The Compiler introduces special record components for certain record type definitions. Such record components are implementation-dependent; they are used by the Compiler to improve the quality of the generated code for certain operations on the record types.

The Compiler issues an error message if you refer to an implementation-dependent component that does not exist. If the implementation-dependent component does exist, the Compiler checks that the storage location specified in the component clause is compatible with the treatment of the component and the storage locations of other components, issuing an error message if this check fails.

Four attributes are defined to refer to these implementation-dependent components in record representation clauses:

T'RECORD_SIZE -- For a prefix T that denotes a record type. This attribute refers to the record component introduced by the Compiler in a record to store the size of the record object. This component exists for objects of a record type with defaulted discriminants when the sizes of the record objects depend on the values of the discriminants.

T'VARIANT_INDEX -- For a prefix T that denotes a record type. This attribute refers to the record component introduced by the Compiler in a record to assist in the efficient implementation of discriminant checks. This component exists for objects of a record type with variant type.

C'ARRAY_DESCRIPTOR -- For a prefix C that denotes a record component of an array type whose component subtype definition depends on discriminants. This attribute refers to the record component introduced by the Compiler in a record to store information on subtypes of components that depend on discriminants.

C'RECORD_DESCRIPTOR -- For a prefix C that denotes a record component of a record type whose component subtype definition depends on discriminants. This attribute refers to the record component introduced by the Compiler in a record to store information on subtypes of components that depend on discriminants.

F.6 ADDRESS CLAUSES

F.6.1 Address Clauses for Objects

As described in the LRM [13.5], you can use an address clause to specify an address for an object. When you do use an address clause, no storage is allocated for the object in the program generated by the Compiler. Instead, the program uses the address specified in the clause to access the object.

You cannot use an address clause for task objects or for unconstrained records whose maximum possible size is greater than 8 Kbytes.

F.6.2 Address Clauses for Program Units

Address clauses for program units are not implemented in the current version of VS Ada.

F.6.3 Address Clauses for Entries

Address clauses for entries are not implemented in the current version of VS Ada.

F.7 Restrictions on Unchecked Conversions

Unconstrained arrays are not allowed as target types, nor are unconstrained record types without defaulted discriminants allowed as target types.

If the source and the target types are each scalar or access types, the sizes of the objects of the source and target types must be equal. If a composite type is used either as the source type or as the target type, this size restriction does not apply.

If the source and the target types are each of scalar or access type or if they are both of composite type, the function returns the operand.

In other cases the effect of unchecked conversion can be considered a copy:

- If an unchecked conversion of a scalar or access source type to a composite target type is achieved, the result of the function is a copy of the source operand; the result is the size of the source.
- If an unchecked conversion of a composite source type to a scalar or access target type is achieved, the result of the function is a copy of the source operand; the result is the size of the target.

F.8 IMPLEMENTATION-DEPENDENT CHARACTERISTICS OF THE INPUT-OUTPUT PACKAGES

The package LOW_LEVEL_IO [14.6], which is concerned with low-level machine-dependent input-output, is not implemented in VS Ada. The predefined input-output packages SEQUENTIAL_IO [14.2.3], DIRECT_IO [14.2.5], TEXT_IO [14.3.10] and IO_EXCEPTIONS [14.5] are implemented as described in the Language Reference Manual.

This section describes those characteristics of the input-out packages that are specific to Wang VS Ada.

F.8.1 Unbounded Line Lengths

VS Ada does not support unbounded line lengths.

F.8.2 The FORM Parameter

The FORM parameter is passed to the Ada CREATE and OPEN procedures to specify VS file attributes.

FORM is an optional parameter. If you omit it or pass it as a null string, the file assumes VS default attributes. If you do include the FORM parameter when creating or opening a file, you need not specify a value for every attribute. In some cases, the attribute may not apply (e.g., a disk file cannot take a tape label type). In other cases, you may choose to accept the default value for the attribute.

The syntax of the FORM string is:

```
form_parameter        ::=  [ attribute [ {, attribute} ] ]

attribute             ::=  key_word [ => value ]
```

Incorrect syntax causes a USE_ERROR exception to be raised.

The following section contains a description of each attribute supported by the FORM parameter.

Attributes of the FORM Parameter

This section lists and describes the VS file attributes supported by the FORM parameter. The attributes are listed alphabetically, and each entry includes the following information:

- Description - A brief description of what the attribute does
- Values - A list of the possible values for the attribute, followed by explanations where necessary
- Default - The default value for the attribute
- Restrictions - Where applicable, a description of any restrictions that apply to the attribute

The *VS Data Management System Reference* contains detailed information about these attributes.

APPEND

Description: The APPEND attribute specifies whether output is appended to the file or overwrites the existing file.

Values: APPEND => [YES | NO]

If APPEND => YES, the record pointer is positioned at end-of-file and output is appended to the file.

If APPEND => NO, the record pointer is positioned at the beginning of the file and output overwrites the existing file.

Default: The default is NO.

Restrictions: Failure to meet the following condition causes a USE_ERROR to be raised:

APPEND => YES can be used only on existing files opened for OUT_FILE or INOUT_FILE mode.

BLOCK_SIZE

Description: The BLOCK_SIZE attribute specifies the size in bytes of the DMS block for the file.

Values: BLOCK_SIZE => [2048 ... 32768]

Default: The default (and only legal value) for DISK, PRINTER, and WS devices is 2048.

The default for TAPE devices is 2048.

BUF_SIZE

Description: The BUF_SIZE attribute specifies the size in bytes of the DMS buffer for the file.

Values: BUF_SIZE => [2048 | 4096 | 6144 | 8192 | 10240 | 12288 | 14336 | 16384 | 18432]

Default: The default is 2048.

Restrictions: Failure to meet the following condition causes a USE_ERROR to be raised:

BUF_SIZE must be a multiple of 2048 bytes (2K), in the range 2048 (2K) to 18432 (18K).

COMPRESS

Description:	The COMPRESS attribute specifies whether or not DMS automatically compresses records.
Values:	COMPRESS => [YES | NO] If COMPRESS => YES, DMS compresses records. If COMPRESS => NO, DMS does not compress records.
Default:	The default is NO.

DENSITY

Description:	The DENSITY attribute applies to tape files only (DEVICE => TAPE). It specifies the density of a tape in bits per inch (BPI).
Values:	DENSITY => [526 | 800 | 1600 | 6250]
Default:	The default is 1600.

DEVICE

Description: The DEVICE attribute specifies the device that is to be associated with the file. The device can be a disk, a magnetic tape, a printer, or a workstation.

Values: DEVICE => [DISK | TAPE | PRINTER | WS]

Default: The default is DISK.

Restrictions: Failure to meet any of the following conditions causes a USE_ERROR to be raised:

If DEVICE => WS, then the file organization must be consecutive (ORGANIZATION => CONSECUTIVE).

If DEVICE => PRINTER, then the file organization must be print (ORGANIZATION => PRINT) and the open access mode must be OUT_FILE.

If DEVICE => TAPE, the file cannot be opened for DIRECT_IO, as DMS allows neither random access to or updating of tape records.

If DEVICE => TAPE, an existing file cannot be opened in OUT_FILE mode, as DMS does not allow updating of tape records.

If DEVICE => DISK, the block size must be 2048 (BLOCK_SIZE => 2048).

DISMOUNT

Description: The DISMOUNT attribute applies to tape files only (DEVICE => TAPE). It specifies whether or not DMS logically dismounts a tape volume when it closes the tape file.

Values: DISMOUNT => [YES | NO]

If DISMOUNT => YES, DMS dismounts the tape.

If DISMOUNT => NO, DMS does not dismount the tape.

Default: The default is NO.

DISPLAY

Description: The DISPLAY attribute specifies whether or not a GETPARM screen is displayed at the workstation at runtime. You can respecify file attributes on the GETPARM screen before opening the file.

Values: DISPLAY => [YES | NO]

If DISPLAY => YES, a GETPARM screen is displayed.

If DISPLAY => NO, a GETPARM screen is not displayed.

Default: The default is NO.

EOF_STRING =>

Description: The EOF_STRING attribute applies to workstation files only (DEVICE = WS). It specifies an end-of-file (eof) string. If a line equal to the eof string is typed in, subsequent calls to the END_OF_FILE function return TRUE; subsequent attempts to read from the workstation raise the END ERROR exception.

Values: EOF_STRING => [/* | sequence_of_characters]

Note: *sequence_of_characters cannot contain commas or spaces.*

Default: The default is /*.

FILE_CLASS

Description: The FILE_CLASS attribute applies to disk files only (DEVICE => DISK). It specifies the file's VS file protection class.

Values: FILE_CLASS => [A ... Z, #, $, @, (blank)]

For information on VS file protection classes, see the *VS System User's Introduction*.

Default: The default is derived from your usage constants.

For information on how to set usage constants, see the *VS System User's Introduction*.

FILE_SEQ

Description:	The FILE_SEQ attribute applies to tape files only (DEVICE => TAPE). It specifies the file sequence number of a tape file.
Values:	FILE_SEQ => [1 ... 9999]
Default:	The default is 1.

FORCE_EOR

Description:	The FORCE_EOR attribute applies to tape files only (DEVICE => TAPE). It specifies whether or not DMS forces an end-of-reel when it closes a tape file that spans multiple volumes.
Values:	FORCE_EOR => [YES | NO] If FORCE_EOR => YES, DMS forces end-of-reel. If FORCE_EOR => NO, DMS does not force end-of-reel.
Default:	The default is NO.

LABEL

Description:	The LABEL attribute applies to tape files only (DEVICE => TAPE). It specifies the label type for a tape.
Values:	LABEL => [NONE | ANY | ANSI | IBM] If LABEL => NONE, the tape contains no labels, or it contains labels that correspond to neither the ANSI nor the IBM standards. DMS treats such as labels as if they were the first data block(s) of the file. If LABEL => ANY, the existing label on the tape is used. If LABEL => ANSI, the tape contains ANSI-standard labels, which are written in ASCII. If LABEL => IBM, the tape contains IBM-standard labels, which are written in EBCDIC.
Default:	The default is ANSI.

NOT_SHARED
SHARED

Description:	The NOT_SHARED/SHARED attribute specifies whether or not several internal files within a single program can share one external file.
Values:	NOT_SHARED SHARED => [READERS | SINGLE_WRITER | ANY] When NOT_SHARED, the external file cannot be shared. When SHARED => READERS, several internal files can read, but not update, the same external file. When SHARED => SINGLE_WRITER, one internal file can update an external file while several other internal files read the same file. When SHARED => ANY, several internal files can both read and update one external file simultaneously.
Default:	The sharing mode is taken from the "brother" files if there are any. In the absence of brother files, if DEVICE => WS, then SHARED => ANY; if MODE => IN_FILE, then SHARED => READERS; otherwise, the sharing mode is NOT_SHARED.

N_RECS

Description:	The N_RECS attribute applies to disk files only (DEVICE => DISK). It specifies your estimate of the number of records the file will contain. DMS uses this estimate to calculate the number of disk blocks to allocate for the primary extent.
Values:	N_RECS => [1 ... 16777215]
Default:	The default is 500.

ORGANIZATION

Description: The ORGANIZATION attribute specifies the file organization type. For further information about file organization, see the *VS Data Management System Reference*.

Values: ORGANIZATION => [CONSECUTIVE | PROGRAM | PRINT]

If ORGANIZATION => CONSECUTIVE, the file is a data file consisting of consecutively written records - that is, records stored in the order in which they are created.

If ORGANIZATION => PROGRAM, the file is a consecutive file of 1024-byte records in VS program format.

If ORGANIZATION => PRINT, the file is a consecutive file containing program output to be sent to a printer.

Default: The default is CONSECUTIVE when DEVICE => DISK, TAPE, and WS. When DEVICE => PRINTER, ORGANIZATION => PRINT.

Restrictions: Failure to meet any of the following conditions causes a USE_ERROR to be raised:

If ORGANIZATION => PROGRAM, then the record size must be 1024 bytes (REC_SIZE => 1024); the record format must be fixed (REC_FORMAT => F); and the file must be opened for either SEQUENTIAL_IO or DIRECT_IO.

If ORGANIZATION => PRINT, then the record format must be variable (REC_FORMAT => V), the records must be compressed (COMPRESS => YES), and the file must be opened for TEXT_IO.

If ORGANIZATION => PRINT, then output must overwrite the existing file (APPEND => NO); since DMS does not allow print files to be opened in I/O mode, output cannot be appended to a print file.

PAD_CHAR

Description: The PAD_CHAR attribute specifies how records are padded.

Values: PAD_CHAR => [NUL | BLANK | any_displayable_ASCII_character]

If PAD_CHAR => NUL, records are padded with nulls (hex 00)

If PAD_CHAR => BLANK, records are padded with blanks (hex 20).

If PAD_CHAR => any_displayable_ASCII_character, records are padded with that character.

Default: The default for sequential and direct files is NUL.

The default for text files is BLANK.

PARITY

Description: The PARITY attribute applies to tape files only (DEVICE => TAPE). It specifies parity for a 7-track tape. (You need not specify parity for 9-track tapes.)

Values: PARITY => [ODD | EVEN]

Default: The default is ODD.

PRINT_CLASS

Description: The PRINT_CLASS attribute applies to print files only (DEVICE = PRINTER). It specifies the file's print class.

Values: PRINT_CLASS => [A ... Z]

For information about print classes, see the *VS System User's Introduction*.

Default: The default is extracted from your usage constants.

For information on how to set usage constants, see the *VS System User's Introduction*.

PRINT_FORM

Description:	The PRINT_FORM attribute applies to print files only (DEVICE = PRINTER). It specifies the type of paper to be mounted in the printer. Printing is then inhibited if the wrong paper is mounted. For further information about form numbers, see the *VS System Operator's Guide*.
Values:	PRINT_FORM => [0 ... 255]
Default:	The default is extracted from your usage constants. For information on how to set usage constants, see the *VS System User's Introduction*.

PR_NAME

Description:	The PR_NAME attribute specifies a parameter reference name (prname) for the file. The prname is used by DMS for GETPARM and PUTPARM processing.
Values:	PR_NAME => [1_to_6_character_string]
Default:	The default is a blank string.

REC_FORMAT

Description:	The REC_FORMAT attribute specifies whether the file contains fixed-length or variable-length records.
Values:	REC_FORMAT => [F | V] IF REC_FORMAT => F, the file contains fixed-length records. If REC_FORMAT => V, the file contains variable-length records.
Default:	The default for sequential and direct files is F. The default for text files is V.

REC_SIZE

Description:	The REC_SIZE attribute specifies the size in bytes of the file's records. For variable length records, REC_SIZE specifies the largest size record that can be written to the file.
Values:	REC_SIZE => [1 ... 2048]
Default:	The default for constrained element types is the size of the instantiated element type. The defaults for unconstrained element types are 2048 for fixed-length records (REC_FORMAT => F); 2024 for variable-length records (REC_FORMAT => V). The default (and only legal value) for program files (ORGANIZATION => PROGRAM) is 1024.
Restrictions:	Failure to meet any of the following conditions causes a USE_ERROR to be raised: If the record format is fixed (REC_FORMAT => F), then REC_SIZE cannot exceed 2048 bytes. If the record format is variable (REC_FORMAT => V), then REC_SIZE cannot exceed 2024 bytes. The record size of files with constrained types that are opened for SEQUENTIAL_IO or DIRECT_IO must be no smaller than ELEMENT_TYPE'SIZE / SYSTEM.STORAGE_UNIT.

RELEASE

Description:	The RELEASE attribute applies to disk files only (DEVICE => DISK). It specifies whether or not DMS returns unused primary extent blocks to the operating system when the file is closed.
Values:	RELEASE => [YES | NO] If RELEASE => YES, DMS returns unused blocks. If RELEASE => NO, DMS does not return unused blocks.
Default:	The default is NO.

RETENTION

Description:	The RETENTION attribute applies to disk files only (DEVICE => DISK). It specifies the number of days beyond the date of its creation that a file is retained.
Values:	RETENTION => [0 ... 999]
Default:	The default is 0 (which specifies that the files's expiration date is the same as its creation date).

REWIND

Description:	The REWIND attribute applies to tape files only (DEVICE => TAPE). It specifies whether or not DMS rewinds the tape volume when it closes the tape file.
Values:	REWIND => [YES | NO]
	If REWIND => YES, DMS rewinds the tape.
	If REWIND => NO, DMS does not rewind the tape.
Default:	The default is NO.

TRACKS

Description:	The TRACKS attribute applies to tape files only (DEVICE => TAPE). It specifies the number of tracks on a tape.
Values:	TRACKS => [7 | 9]
Default:	The default is 9.

TRUNCATE

Description:	The TRUNCATE attribute specifies whether or not records in the file are truncated by eliminating trailing blanks. The TRUNCATE attribute applies only to files opened for TEXT_IO in IN_FILE mode.
Values:	TRUNCATE => [YES | NO]
	If TRUNCATE => YES, records are truncated.
	If TRUNCATE => NO, records are not truncated.
Default:	The default is NO.

VOL_NUM

Description:	The VOL_NUM attribute applies to tape files only (DEVICE => TAPE). It specifies the volume sequence number of a tape. Volume sequence numbers are used when a file spans several tape volumes.
Values:	VOL_NUM => [1 ... 9999]
Default:	The default is 1.

Examples of FORM Parameter Usage

The following examples illustrate the FORM parameter as it appears in calls to the CREATE or OPEN procedures:

Example 1

```
CREATE( FILE => FD,
        MODE => OUT_FILE,
        NAME => "FORT77.JM1#SRCE.TEST",
        FORM => "DEVICE => DISK, ORGANIZATION => PROGRAM,
                 REC_SIZE => 1024, REC_FORMAT => F" );
```

This example creates a program file. Note that the DEVICE, REC_SIZE, and REC_FORMAT attributes could be omitted, since the values supplied for those attributes match their default values.

Example 2

```
OPEN(   FILE => FD,
        MODE => IN_FILE,
        NAME => "TAPE.JM1#SRCE.TEST",
        FORM => "DEVICE => TAPE, REC_SIZE => 80, DENSITY => 6250,
                 LABEL => NONE" );
```

This example opens a 6250-bpi, nonlabeled, nine-track tape file in read-only mode. The tape contains 80-byte, fixed-length, consecutive records.

Example 3

```
CREATE( FILE => FD,
        MODE => OUT_FILE,
        NAME => "FORT77.#JM1PRT.PRTFILE",
        FORM => "ORGANIZATION => PRINT, REC_SIZE => 134,
                 COMPRESS => YES, BUF_SIZE => 18432" );
```

This example creates a print file containing 134-byte print records. The BUF_SIZE attribute specifies an 18K DMS buffer to maximize performance.

F.9 CHARACTERISTICS OF NUMERIC TYPES

F.9.1 Integer Types

The ranges of values for integer types declared in package STANDARD are as follows:

```
SHORT_SHORT_INTEGER   -128 .. 127 -- -2**7 .. 2**7 - 1

SHORT_INTEGER         -32768 .. 32767 -- -2**15 .. 2**15 - 1

INTEGER               -2147483648 .. 2147483647  -- -2**31 .. 2**31 - 1
```

The ranges of values for types COUNT and POSITIVE_COUNT declared in packages DIRECT_IO and TEXT_IO are as follows:

```
COUNT                 0 .. 2147483647                -- 0 .. 2**31 -1

POSITIVE_COUNT        1 .. 2147483647                -- 1 .. 2**31 -1
```

The range of values for the type FIELD declared in package TEXT_IO is as follows:

```
FIELD                 0 .. 255                       -- 0 .. 2**8 -1
```

F.9.2 Floating Point Type Attributes

SHORT_FLOAT

		Approximate Value
DIGITS	6	
MANTISSA	21	
EMAX	84	
EPSILON	2.0 ** -20	9.54E-07
SMALL	2.0 ** -85	2.58E-26
LARGE	2.0 ** 84 * (1.0 -2.0 ** -21)	1.93E+25
SAFE_EMAX	252	
SAFE_SMALL	2.0 ** -253	6.91E-77
SAFE_LARGE	2.0 ** 252 * (1.0 - 2.0 ** -21)	7.24E+75
FIRST	-2.0 ** 252 * (1.0 - 2.0 ** -24)	-7.24E+75
LAST	2.0 ** 252 * (1.0 - 2.0 ** -24)	7.24E+75
MACHINE_RADIX	16	
MACHINE_MANTISSA	6	
MACHINE_EMAX	63	
MACHINE_EMIN	-64	
MACHINE_ROUNDS	FALSE	
MACHINE_OVERFLOWS	TRUE	
SIZE	32	

FLOAT

		Approximate Value
DIGITS	15	
MANTISSA	51	
EMAX	204	
EPSILON	2.0 ** -50	8.88E-16
SMALL	2.0 ** -205	1.94E-62
LARGE	2.0 ** 204 * (1.0 - 2.0 ** -51)	2.57E+61
SAFE_EMAX	252	
SAFE_SMALL	2.0 ** -253	6.91E-77
SAFE_LARGE	2.0 ** 252 * (1.0 - 2.0 ** -51)	7.24E+75
FIRST	-2.0 ** 252 * (1.0 - 2.0 ** -56)	-7.24E+75
LAST	2.0 ** 252 * (1.0 - 2.0 ** -56)	7.24E+75
MACHINE_RADIX	16	
MACHINE_MANTISSA	14	
MACHINE_EMAX	63	
MACHINE_EMIN	-64	
MACHINE_ROUNDS	FALSE	
MACHINE_OVERFLOWS	TRUE	
SIZE	64	

F.9.3 Attributes of Type DURATION

DURATION'DELTA	2.0 ** -6
DURATION'SMALL	2.0 ** -6
DURATION'LARGE	131072.0
DURATION'FIRST	-86400.0
DURATION'LAST	86400.0

F.10 OTHER IMPLEMENTATION-DEPENDENT CHARACTERISTICS

F.10.1 Characteristics of the Heap

All objects created by allocators go into the program heap. In addition, portions of the Ada runtime system's representation of task objects, including the task stacks, are allocated in the program heap.

All objects in the heap belonging to a given collection have their storage reclaimed upon exit from the innermost block statement, subprogram body, or task body that encloses the access type declaration associated with the collection. For access types declared at the library level, this deallocation occurs only upon completion of the main program.

No further automatic storage reclamation is performed; i.e. in effect, all access types are deemed to be controlled [4.8]. You can achieve explicit deallocation of the object designated by an access value by calling an appropriate instantiation of the generic procedure UNCHECKED_DEALLOCATION.

Space for the heap is initially claimed from the system at program start-up, and additional space may be claimed as required when the initial allocation is exhausted. You can use the Binder options HEAP and MOREHEAP to control both the size of the initial allocation and the size of the individual increments. You can also use ADAPATCH to set these values.

F.10.2 Characteristics of Tasks

The default initial task stack size is 16 Kbytes. Using either the Binder option TASK or the ADAPATCH program, you can set all task stacks in a program to any size from 4 Kbytes to 16 Mbytes.

The maximum number of active tasks is limited only by memory usage. Tasks release their storage allocation as soon as they have terminated.

The acceptor of a rendezvous executes the accept body code in its own stack. A rendezvous with an empty accept body (e.g. for synchronization) need not cause a context switch.

The main program waits for completion of all tasks dependent on library packages before terminating. Such tasks may select a terminate alternative only after completion of the main program.

Abnormal completion of an aborted task takes place immediately unless the abnormal task is the caller of an entry that is engaged in a rendezvous. In this case, abnormal completion of the task takes place as soon as the rendezvous is completed.

A global deadlock situation arises when every task, including the main program, is waiting for another task. When this happens, the program is aborted and the state of all tasks is displayed.

F.10.3 Definition of a Main Program

A main program must be a nongeneric library procedure with no parameters.

F.10.4 Ordering of Compilation Units

The VS Ada Compiler imposes no ordering constraints on compilations beyond those required by the language.

F.10.5 Implementation-Defined Packages

The following packages are defined by the VS Ada implementation.

Package SYSTEM_ENVIRONMENT

The VS-defined package SYSTEM_ENVIRONMENT enables an Ada program to communicate with the environment in which it is executed.

The specification of package SYSTEM_ENVIRONMENT is as follows:

```
package SYSTEM_ENVIRONMENT is

   subtype EXIT_STATUS is INTEGER:

   procedure SET_EXIT_STATUS (STATUS : in EXIT_STATUS);

   procedure ABORT_PROGRAM (STATUS : in EXIT_STATUS);

   procedure CLEAR_SCREEN;

end SYSTEM_ENVIRONMENT;
```

SET_EXIT_STATUS

You can set the exit status of the program (returned in register 0 on exit) by calling SET_EXIT_STATUS. Subsequent calls of SET_EXIT_STATUS modify the exit status; the status finally returned is the status specified by the last executed call to SET_EXIT_STATUS. If SET_EXIT_STATUS is not called, the value 0 is returned.

ABORT_PROGRAM

You can cause the program to be aborted, returning the specified exit code, by calling the ABORT_PROGRAM procedure.

CLEAR_SCREEN

You can erase the entire contents of the workstation screen by calling the CLEAR_SCREEN procedure.

Package STRINGS

The VS Ada-defined package STRINGS is a utility package that provides many commonly required string manipulation facilities.

The specification of package STRINGS is as follows:

```
with UNCHECKED_DEALLOCATION;
package STRINGS is

   --                     *********
   --                     * TYPES *
   --                     *********

   type ACCESS_STRING is access STRING;
   procedure DEALLOCATE_STRING is new UNCHECKED_DEALLOCATION (STRING,
                                        ACCESS_STRING);
```

```
    --          **************
    --          * UTILITIES *
    --          **************

    function  UPPER (C : in CHARACTER) return CHARACTER;
    function  UPPER (S : in STRING) return STRING;
    procedure UPPER (S : in out STRING);

    function  LOWER (C : in CHARACTER) return CHARACTER;
    function  LOWER (S : in STRING) return STRING;
    procedure LOWER (S : in out STRING);

    function  CAPITAL (S : in STRING) return STRING;
    procedure CAPITAL (S : in out STRING);

    function  REMOVE_LEADING_BLANKS   (S : in STRING) return STRING;
    function  REMOVE_TRAILING_BLANKS  (S : in STRING) return STRING;
    function  TRIM (S : in STRING) return STRING;

    function  INDEX  (C       : in CHARACTER;
                      INTO    : in STRING;
                      START   : in POSITIVE := 1) return NATURAL;
    function  INDEX  (S       : in STRING;
                      INTO    : in STRING;
                      START   : in POSITIVE  := 1) return NATURAL;

    function NOT_INDEX (C     : in CHARACTER;
                        INTO  : in STRING;
                        START : in POSITIVE := 1) return NATURAL;
    function NOT_INDEX (S     : in STRING;
                        INTO  : in STRING;
                        START : in POSITIVE := 1) return NATURAL;

    function IS_AN_ABBREV (ABBREV      : in STRING;
                           FULL_WORD   : in STRING;
                           IGNORE_CASE : in BOOLEAN := TRUE) return BOOLEAN

    function MATCH_PATTERN (S           : in STRING;
                            PATTERN     : in STRING;
                            IGNORE_CASE: in BOOLEAN := TRUE) return BOOLEAN

    function '&' (LEFT : in STRING; RIGHT : in STRING) return STRING;
    function '&' (LEFT : in STRING; RIGHT : in CHARACTER) return STRING;
    function '&' (LEFT : in CHARACTER; RIGHT : in STRING) return STRING;
    function '&' (LEFT : in CHARACTER; RIGHT : in CHARACTER) return STRING;

end STRINGS;
```

ACCESS_STRING

The ACCESS_STRING type is a convenient declaration of the commonly used access to string type.

DEALLOCATE_STRING

The DEALLOCATE_STRING procedure is an instantiation of UNCHECKED_DEALLOCATION for the type ACCESS_STRING. Note that since the type ACCESS_STRING is declared at the library level, the scope of the corresponding collection is exited only at program completion. For this reason, STRING objects belonging to this collection are never automatically deallocated. It is therefore your responsibility to manage the deallocation of objects within this collection.

UPPER

The UPPER subprograms convert any lower case-letters in their parameters to the corresponding upper-case letters. Characters that are not lower-case letters are not affected. The procedure is more efficient than the corresponding function, as it does not use the program heap.

LOWER

The LOWER subprograms convert any upper-case letters in their parameters to the corresponding lower-case letters. Characters that are not upper-case letters are not affected. The procedure is more efficient than the corresponding function, as it does not use the program heap.

CAPITAL

The CAPITAL subprograms "capitalize" their parameters. That is, they convert the first character of the string to upper case and all subsequent characters to lower case. The procedure is more efficient than the corresponding function, as it does not use the program heap.

REMOVE_LEADING_BLANKS

The REMOVE_LEADING_BLANKS function returns its parameter string with all leading spaces removed.

REMOVE_TRAILING_BLANKS

The REMOVE_TRAILING_BLANKS function returns its parameter string with all trailing spaces removed.

TRIM

The TRIM function returns its parameter string with all leading and trailing spaces removed.

INDEX

The INDEX subprograms return the index into the specified string (INTO) of the first character of the first occurrence of a given substring (S) or character (C). The search for the substring or character commences at the index specified by START. If the substring or character is not found, the functions return the value 0. Case is significant.

NOT_INDEX

The NOT_INDEX subprograms return the index into the specified string (INTO) of the first character that does not occur in the given string (S) or does not match the given character (C). The search for the nonmatching character commences at the index specified by START. If all the characters of the string match, the functions return the value 0. Case is significant.

IS_AN_ABBREV

The IS_AN_ABBREV function determines whether the string ABBREV is an abbreviation for the string FULL_WORD. Leading and trailing spaces in ABBREV are first removed, and the trimmed string is then considered to be an abbreviation for FULL_WORD if it is a proper prefix of FULL_WORD.

The parameter IGNORE_CASE controls whether or not case is significant.

MATCH_PATTERN

The MATCH_PATTERN function determines whether the string S matches the pattern specified in PATTERN. A pattern is simply a string in which the character '*' is considered a wild-card that can match any number of any characters.

For example, the string "ABCDEFG" matches the pattern "A*G" and the pattern "ABCD*EFG*".

The parameter IGNORE_CASE controls whether or not case is significant.

The package STRINGS also provides overloaded subprograms designated by '&'. These are identical to the corresponding subprograms declared in package STANDARD, except that the concatenations are performed out of line. Performing concatenations out of line minimizes the size of the inline generated code, at the expense of execution speed.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	(1..254 => 'A', 255 => '1')
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	(1..254 => 'A', 255 => '2')
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	(1..127 => 'A', 128 => '3', 129..255 => 'A')

TEST PARAMETERS

Name and Meaning	Value
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	(1..127 => 'A', 128 => '4', 129..255 => 'A')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..252 => '0', 253..255 => "298")
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..250 => '0', 251..255 => "690.0")
$BIG_STRING1 A string literal which when catenated with $BIG_STRING2 yields the image of $BIG_ID1.	(1 => '"', 2..129 => 'A', 130 => '"')
$BIG_STRING2 A string literal which when catenated to the end of $BIG_STRING1 yields the image of $BIG_ID1.	(1 => '"', 2..127 => 'A', 128 => '1', 129 => '"')
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..235 => ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	16777216
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	WANG_VS
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	255
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	10000000.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	1
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	BAD-FNAM
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	FILE#NAME#TOO#LONG
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648

TEST PARAMETERS

Name and Meaning	Value
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-10000000.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	1
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	255
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be $MAX_IN_LEN long.	(1..2 => "2:", 3..252 => '0', 253..255 => "11:")

Name and Meaning	Value
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be $MAX_IN_LEN long.	(1..3 => "16:", 4..251 => '0', 252..255 => "F.E:")
$MAX_STRING_LITERAL A string literal of size $MAX_IN_LEN, including the quote characters.	(1 => '"', 2..254 => 'A', 255 => '"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	SHORT_SHORT_INTEGER
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	WANG_VS
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFF#
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	0

TEST PARAMETERS

Name and Meaning	Value
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	0
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	WANG_VS
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	0.01

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C: This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this text that must appear at the top of the page.

b. A39005G: This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

c. B97102E: This test contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

d. C97116A: This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

e. BC3009B: This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

f. CD2A62D: This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, and CD2A76A..D (16 tests): These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

h. CD2A81G, CD2A83G, CD2A84M..N, and CD50110 (5 tests): These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86, 96, and 58, respectively).

i. CD2B15C and CD7205C: These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

j. CD2D11B: This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

k. CD5007B: This test wrongly expects an implicitly declared subprogram to be at the address that is specified for an unrelated subprogram (line 303).

l. ED7004B, ED7005C..D, and ED7006C..D (5 tests): These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

m. CD7105A: This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

n. CD7203B and CD7204B: These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

o. CD7205D: This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

p. CE2107I: This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid (line 90).

q. CE3111C: This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A: This test contains several calls to END_OF_LINE and END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, and 136).

s. CE3411B: This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER OPTIONS AS SUPPLIED BY WANG

Compiler:	Wang VS Ada, Version 4.1
ACVC Version:	1.10

COMPILER OPTIONS

The default option is either underlined or in parenthesis

Option	Effect
DEBUG=YES/NO	Specifies whether or not symbolic debugger information is generated.
LEVEL=PARSE/SEMANTIC/CODE/OBJECT	Specifies the level to which the compiler analyzes the source code.
GENERICS=YES/NO	Specifies whether or not code for generics is inlined.
ERRORS=1-999 (50)	Specifies the maximum number of compilation errors permitted (compilation is terminated if this number is exceeded).
MEMORY=256-4095 (500)	Specifies the number of kilobytes reserved in memory for program library data.
ANNOTATE=YES/NO	Specifies whether comments (entered on a subsequent menu) should be saved in the program library with the unit being compiled.
CHECKS=ALL/STACK	Specifies the level of runtime checking.
INLINE=YES/NO	Specifies whether code is inlined.
OPTIMIZE=YES/NO	Specifies whether generated code is optimized.
PEEPHOLE=YES/NO	Specifies whether peephole optimizations are performed.
SOURCE=YES/NO	Specifies whether source text is included in the compilation listing.
DMAP=YES/NO	Specifies whether a data map is included in the compilation listing.
PMAP=YES/NO	Specifies whether a program map is included in the compilation listing.
WARNING=YES/NO	Specifies whether warning messages are included in the compilation listing.
DETAIL=YES/NO	Specifies whether extra informational detail is included in the compilation listing.
LINELEN=80-132 (132)	Specifies the width in characters of the listing file.
BANNER=YES/NO	Specifies whether banners are included in the compilation listing.
STACK=256-4095 (1024)	Controls the allocation of stack objects.
GLOBAL=256-4095 (1024)	Controls the allocation of global objects.
UNNESTED=16-4095 (16)	Controls the allocation of unnested objects.